THE UNIVERSITY OF WESTERN AUSTRALIA

Core JavaScript: Objects, and Functions

CITS3403 Agile Web Develpopment

Semester 1, 2018

Object Orientation and JavaScript

THE UNIVERSITY OF WESTERN AUSTRALIA

- JavaScript is *object-based*
  - JavaScript defines objects that encapsulate both data and processing
  - However, JavaScript does not have the same inheritance nor subtyping (therefore polymorphism) as normal OOP such as Java or C#.
- JavaScript provides *prototype-based inheritance*
  - See, for example this Wikipedia article for a discussion: http://en.wikipedia.org/wiki/Prototype-based_languages

- Objects are collections of *properties*
- Properties are either *data properties* or *method properties*
  - Data properties are either *primitive values* or *references to other objects*
  - Primitive values are often implemented directly in hardware
  - Method properties are *functions* (more later)
- The *Object* object is the ancestor of all objects in a JavaScript program
  - Object has no data properties, but several method properties

Arrays

- Arrays are lists of elements indexed by a numerical value
- Array indexes in JavaScript begin at 0
- *Arrays can be modified in size even after they have been created*

- *E.g.*

```
var index;
var fruits = ["Banana", "Orange", "Apple", "Mango"];
for (index = 0; index < fruits.length; index++) {
    text += fruits[index];
}
```

```
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
var x = person.length;         // person.length will return 3
var y = person[0];             // person[0] will return "John"
```

`Array` Object Creation

THE UNIVERSITY OF WESTERN AUSTRALIA

- Arrays can be created using the `new Array` method
  - new Array with one parameter creates an empty array of the specified number of elements

    ```
    new Array(10);
    ```
  - new Array with no parameter creates an empty array

    ```
    var a = new Array();
    a[0] = "dog"; a[1] = "cat"; a[2] = "hen";
    console.log(a.length);  // outputs 3
    ```
  - new Array with two or more parameters creates an array with the specified parameters as elements

    ```
    new Array(1, 2, "three", "four");
    ```

- Literal arrays can be specified using square brackets to include a list of elements

  ```
  var alist = [1, "ii", "gamma", "4"];
  ```

- It is better to avoid the "new" keyword where possible
- Elements of an array *do not have to be of the same type*

Characteristics of Array Objects

THE UNIVERSITY OF WESTERN AUSTRALIA

- The length of an array is one more than the highest index
- You can iterate over an array using this length property, or you can use for….in construct

```
for(var i in a)
console.log( a[i] );
```

- Assignment to an index greater than or equal to the current length simply increases the length of the array
  - `a[100] = "lion";  console.log(a.length);`
  - (Note: errors may go unnoticed.)
- Only assigned elements of an array occupy space
  - Suppose an array were created using new Array(200)
  - Suppose only elements 150 through 174 were assigned values
  - Only the 25 assigned elements would be allocated storage, the other 175 would not be allocated storage
- If you query a non-existent array index, you get undefined –

```
console.log(a[90])          // outputs undefined
```

`Array` Methods

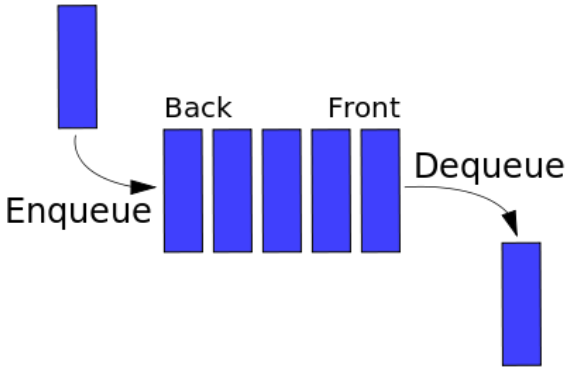
- join — *returns a string of the elements in the array*
- reverse — *….reverses the array*
- sort — *…. sorts the array, can take a comparator function as an argument*
- concat — *concatenates 2 or more arrays*
- slice — *creates 2 arrays from 1 array*
- splice — *inserts a group of elements at a given index*
- delete — *replaces an element at an index with undefined*

Associative Arrays index on Strings and are actually Objects. These oeprations are not available to them:

```
var arr = [];
arr[“name”] = “Bob”;
```

Dynamic List Operations

- push
  - Add to the end
- pop
  - Remove from the end
- shift
  - Remove from the front
- unshift
  - Add to the front



Like Perl and Ruby, Javascript already have operations for pushing and popping an array from both ends, so one can use **push** and **shift** functions to enqueue and dequeue a list (or, in reverse, one can use **unshift** and **pop**)

- A two-dimensional array in JavaScript is an array of arrays
  - This need not even be rectangular shaped: different rows could have different length
- Example `nested_arrays.js` illustrates two-dimensional arrays

Function Fundamentals

THE UNIVERSITY OF WESTERN AUSTRALIA

- Function definition syntax
  - A function definition consists of a header followed by a compound statement
  - A function header:
    - *function function-name(optional-formal-parameters)*
- Function call syntax
  - Function name followed by parentheses and any actual parameters
  - Function call may be used as an expression or part of an expression
- Functions must be defined before use in the page header (or linked in an external file)

- return statements
  - A return statement causes a function to cease execution and control to pass to the caller
  - A return statement may include a value which is sent back to the caller
  - If the function doesn't have any return statement, or uses an empty return with no value, then *undefined* is returned.

Functions

- Along with the objects, functions are the core components in understanding Javascript. We can also treat functions as objects. The most basic function is as follows

```
function add(x, y){
        var total = x+y;
        return total;
}
```

- You can call the above function with no parameter as well. In such case, they will be set to *undefined*.

Calling Functions from HTML

- JavaScript file:

```
function myfunction (myparameter1, myparameter 2, ...)
{
  // do something
  console.log("My answer is...",answer); // or maybe...
  return answer;
}
```

- XHTML file:

```
<head>
   <script src="somefile.js"></script>
</head>
<body>
  <script>
     myfunction(7,6);
  </script>
</body>
```

Functions are Objects

- *Functions are objects* in JavaScript (or *first class functions*)
- Functions may, therefore, be assigned to variables and to object properties
  - *Object properties that have function name as values are methods of the object*

Example

```
function fun() {
    console.log("This surely is fun!");
}
ref_fun = fun; // Now, ref_fun refers to
               // the fun object
fun();          // A call to fun
ref_fun();      // Also a call to fun
```

Local Variables

- "The *scope* of a variable is the range of statements over which it is visible"
- *A variable not declared using **var** has global scope*, visible throughout the page, even if used inside a function definition
- A variable declared with ***var*** outside a function definition has global scope
- A variable declared with ***var*** inside a function definition has local scope, visible only inside the function definition
  - If a global variable has the same name, it is hidden inside the function definition

Parameters

- Parameters named in a function header are called ***formal parameters***
- Parameters used in a function call are called ***actual parameters***
- Use arguments to access non-formal parameters

```
x = findMax(1, 123, 500, 115, 44, 88);

function findMax() {
    var i;
    var max = -Infinity;
    for (i = 0; i < arguments.length; i++) {
        if (arguments[i] > max) {
            max = arguments[i];
        }
    }
    return max;
}
```

Parameters are *passed by value*
For an object parameter, the reference is passed, so the function body can actually change the object (effectively *pass by reference*)
However, an assignment to the formal parameter will not change the actual parameter

Parameter Passing Example

```
function fun1(my_list) {
  var list2 = new Array(1, 3, 5);
  my_list[3] = 14; //changes actual parameter

  my_list = list2; //no effect on actual parameter

  return my_list;
}

var list = new Array(2, 4, 6, 8)
fun1(list);
```

- The first assignment changes list in the caller
- The second assignment has no effect on the list object in the caller

Parameter Checking

- JavaScript checks neither the type nor number of parameters in a function call
  - Formal parameters have no type specified
  - Extra actual parameters are ignored (however, see below)
  - If there are fewer actual parameters than formal parameters, the extra formal parameters remain undefined
- This flexibility is typical of many scripting languages
  - different numbers of parameters may be appropriate for different uses of the function
- A property array named arguments holds all of the actual parameters, whether or not there are more of them than there are formal parameters

Functions

- You can pass in more arguments than the function is expecting

```
console.log( add(2, 3, 4) ); // outputs 5
```

- Functions have access to an additional variable inside their body called *arguments*, which is an array-like objects holding all of the values passed to that function. Let's write a function which takes as many arguments values as we want

```
function avg(){
      var sum = 0;
      for (var i=0; i<arguments.length; i++)
                  sum += arguments[i];
      return sum / arguments.length;
}
```

Functions

- What if we want to calculate the average value of an array? We can re-use the above function for arrays in the following way

```
console.log( avg.apply(null, [2, 3, 4, 5]) );      // outputs 3.5

// apply() has a sister function called call
var x = 10;  var o = { x: 15 };
function f(message) {
    console.log(message, this.x);
}
f("invoking f");
f.call(o, "invoking f via call");
```

- In Javascript, you can create anonymous functions

```
var avg = function() {  // the rest of the body.....  }
```

This is extremely powerful as it lets you put a function definition anywhere that you would normally put an expression.

Function Passing Example
The `sort` Method

- A parameter can be passed to the sort method to specify how to sort elements in an array
  - The parameter is a function that takes two parameters
  - The function returns a negative value to indicate the first parameter should come before the second
  - The function returns a positive value to indicate the first parameter should come after the second
  - The function returns 0 to indicate the first parameter and the second parameter are equivalent as far as the ordering is concerned
- Example:

```
var points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return b>a});
```

Constructors

- Constructors are functions that create and initialize properties for new objects
- A constructor uses the keyword `this` in the body to reference the object being initialized
- Object methods are properties that refer to functions
  - A function to be used as a method may use the keyword `this` to refer to the object for which it is acting
- Example `car_constructor.js`

Functions (Recursive)

- Like any other languages, you can write recursive functions in Javascript. However, this creates a problem if the function is anonymous. How would you call a function without its name? The solution is using *named anonymous functions* -

```
var ninja = {
        yell: function cry(n) {
                return n > 0 ? cry(n-1) + "a" : "hiy";
        }
};

console.log( ninja.yell(5) );       // outputs hiyaaaaa
```

Built-in Objects

The Math Object

THE UNIVERSITY OF WESTERN AUSTRALIA

- Provides a collection of properties and methods useful for Number values
- This includes the trigonometric functions such as `sin` and `cos`
- When used, the methods must be qualified, as in `Math.sin(x)`
- See http://www.w3schools.com/js/js_math.asp

The Date Object

- A Date object represents a *time stamp*, that is, a point in time
- A Date object is created with the new operator
  - `var d = new Date();`
  - creates a Date object for the time at which it was created
    - `d.setFullYear(2003,10,5);`
  - resets to 5th November 2003

The `Date` Object: Methods

THE UNIVERSITY OF WESTERN AUSTRALIA

Method	Returns
`toLocaleString`	A string of the `Date` information
`getDate`	The day of the month
`getMonth`	The month of the year, as a number in the range of 0 to 11
`getDay`	The day of the week, as a number in the range of 0 to 6
`getFullYear`	The year
`getTime`	The number of milliseconds since January 1, 1970
`getHours`	The number of the hour, as a number in the range of 0 to 23
`getMinutes`	The number of the minute, as a number in the range of 0 to 59
`getSeconds`	The number of the second, as a number in the range of 0 to 59
`getMilliseconds`	The number of the millisecond, as a number in the range of 0 to 999

Window and Document

- The Window object represents the window in which the document containing the script is being displayed
- The Document object represents the document being displayed using DOM (more on this later...)
- Window has two properties
  - `window` refers to the Window object itself
  - `document` refers to the Document object
- The Window object is the default object for JavaScript, so properties and methods of the Window object may be used without qualifying with the class name

Javascript IO

- Standard output for JavaScript embedded in a browser is the window displaying the page in which the JavaScript is embedded
- Writing to the document object is now considered bad practice. For simple debugging use

```
console.log("The result is: ", result, "<br />");
```

- To read, you can use *alert* or *confirm.* To get input you can use *prompt.*
- In NodeJS you can access `stdin, stdout`, and `stderr` through the *process* object. Eg:

```
const readline = require('readline');
const rl = readline.createInterface({
      input: process.stdin,
      output: process.stdout});
rl.question('What do you think of Node.js? ', (answer) =>
{
      console.log('Thank you for your feedback:', answer);
      rl.close();});
```

Errors in Scripts

- JavaScript errors are detected by the browser
- Different browsers report this differently
  - Firefox uses a special console
- Can insert breakpoint in code with:

```
debugger;
```

- Support for debugging is provided
  - IE, the debugger is part of the browser
  - Firefox , plug-ins are available
    - These include Venkman and Firebug
  - Safari: Develop | Show Error Console
    - First use: Choose Preferences | Advanced | Show Develop menu in menu bar
    - Note: Reopen error console after reloading page (bug?)
  - Chrome
    - Use console from the Developer Tools

User-Defined Objects

THE UNIVERSITY OF WESTERN AUSTRALIA

- JavaScript objects are simply collections of name-value pairs. As such, they are similar to HashMaps in Java.
- An object may be thought of as a Map/Dictionary/Associative-Storage.
- If a variable is not a primitive (undefined, null, boolean, number or string), its an object.
- The name part is a string, while the value can be any JavaScript value – including objects.

Object Creation and Modification

THE UNIVERSITY OF WESTERN AUSTRALIA

- There are two basic ways to create an empty object –
- The `new` expression is used to create an object

```
var obj = new Object();
```

  - This includes a call to a *constructor*
  - The new operator creates a blank object, the constructor creates and initializes all properties of the object
- The second is called object literal syntax. It's also the core of JSON format and should be preferred at all times.

```
// sets the objects prototype to Object.prototype
var obj = {};
// sets null as object prototype
var obj = Object.create(null);
```

Object literal

- Object literal syntax can be used to initialize an object in its entirety –

```
var obj = {
            name: "Carrot",
            for : "Max",
            detail: { color: "Orange", size: 12 }
          };
```

- Attribute access can be chained together –

```
console.log(obj.detail.color);
```

Accessing Object Properties

THE UNIVERSITY OF WESTERN AUSTRALIA

- Just like Java, an object's properties can be accessed using the dot operator -
  - `Unit.name = "Agile Web Programming"`
- And using the array-like index –
  - `Unit["name"] = "Agile Web Programming";`
- Both of these methods are semantically equivalent.
- The second method has the advantage that the name of the property is provided as a string, which means it can be calculated at run-time. It can also be used to set and get properties with names that are reserved words.

Dynamic Properties

THE UNIVERSITY OF WESTERN AUSTRALIA

- Create my_car and add some properties

```
// Create an Object object
var my_car = new Object();
// Create and initialize the make property
my_car.make = "Ford";
// Create and initialize model
my_car.model = "Contour SVT";
```

- The delete operator can be used to delete a property from an object
  - `delete my_car.model`

The *for-in* Loop

- Syntax

  for (*identifier* in *object*)

  statement or compound statement

- The loop lets the identifier take on each property in turn in the object

```
for (var prop in my_car)
   console.log("Key: ", prop, "; Value:",my_car[prop]);
```

- Result:
  - `Name: make; Value: Ford`
  - `Name: model; Value: Contour SVT`

Creating Object Properties

```
var person = Object.create(null);
Object.defineProperty(person, 'firstName', {
        value: "Yehuda", writable: true, enumerable: true,
                                        configurable: true
});

Object.defineProperty(person, 'lastName', {
        value: "Katz", writable: true, enumerable: true,
        configurable: true
});
```

Object-orientation in JavaScript

THE UNIVERSITY OF WESTERN AUSTRALIA

- JavaScript doesn't have classes, so its object-oriented approach doesn't match that of other popular OOP languages like Java, C# etc. Instead, it supports a variation of Object-oriented programming known as Prototype-based Programming.
- In prototype-based programming, classes are not present, and behavior reuse (equivalent to *inheritance* in Java) is accomplished through a process of decorating existing objects which serves as prototypes. This model is also known as *class-less*, *prototype-oriented* or *instance-based programming.*
- Just like Java, every object in Javascript is an instance of the object *Object* and therefore inherits all its properties and methods.

The *this* keyword

THE UNIVERSITY OF WESTERN AUSTRALIA

- When used inside a function, this refers to the current object. What that actually means is specified by the way in which you called that function.
- In the global scope of a browser it refers to the window displaying the HTML.
- In Node, it refers to the execution environment.
- If you called it using the dot notation or bracket notation on an object, that object becomes this. If any of these notations wasn't used for the call, then this refers to the global object (the window object). For example

```
s = makePerson("Simon", "Willison")
var fullName = s.fullName;
console.log( fullName() );    // will output undefined
```

Using *this* for objects

- We can take advantage of *this* keyword to improve our function in the following way

```
function Person(first, last) {
        this.first = first;
        this.last = last;
        this.fullName = function() {
           return this.first + ' ' + this.last;
        }
        this.fullNameReversed = function() {
           return this.last + ', ' + this.first;
        }
}

var s = new Person("Kowser Vai", "the Ice-cream Guy");
```

The *new* keyword

THE UNIVERSITY OF WESTERN AUSTRALIA

- new is strongly related to this. What it does is it creates a brand new empty object, and then calls the function specified, with this set to that new object. Functions that are designed to be called by new are called constructor functions.
- When the code new Person(…) is executed, the following things happen:
  1. A new object is created, inheriting from Person.prototype.
  2. The constructor function Person is called with the specified arguments and this bound to the newly created object. new Person is equivalent to new Person (), i.e. if no argument list is specified, Person is called without arguments.
  3. The object returned by the constructor function becomes the result of the whole new expression. If the constructor function doesn't explicitly return an object, the object created in step 1 is used instead. (Normally constructors don't return a value, but they can choose to do so if they want to override the normal object creation process.)

Function objects reuse

- Every time we are creating a *person* object, we are creating two new brand new function objects within it. Wouldn't it be better if this code was shared? There are two ways in which code can be shared. The first way is the following

```
function personFullName() {
    return this.first + ' ' + this.last;
}
function personFullNameReversed() {
    return this.last + ', ' + this.first;
}
function Person(first, last) {
    this.first = first;
    this.last = last;
    this.fullName = personFullName;
    this.fullNameReversed = personFullNameReversed;
}
```

Function objects reuse

- The second (and best) way is to use the *prototype*

```
function Person(first, last) {
    this.first = first;
    this.last = last;
}
Person.prototype.fullName = function() {
    return this.first + ' ' + this.last;
}
Person.prototype.fullNameReversed = function() {
    return this.last + ', ' + this.first;
}
```

The *prototype*

THE UNIVERSITY OF WESTERN AUSTRALIA

- *Person.prototype* is an object shared by all instances of Person. It forms a part of a lookup chain (or, *prototype chain*) : any time you attempt to access a property of *Person* that isn't set, Javascript will check *Person.prototype* to see if that property exists there instead. As a result, anything assigned to *Person.prototype* becomes available to all instances of that constructor via the *this* object. The root of the prototype chain is *Object.prototype*.
- This is an incredibly powerful tool. Javascript lets you modify something's prototype at anytime in your program, which means you can add extra methods to existing objects at runtime.

Adding methods at run time using prototype

THE UNIVERSITY OF WESTERN AUSTRALIA

```
var s = “Issa";
String.prototype.reversed = function(){
    var r = "";
    for (var i = this.length - 1; i >= 0; i--){
        r += this[i];
    }
    return r;
}
s.reversed();           // will output assi

“This can now be reversed".reversed()
// outputs desrever eb won nac sihT
```

JavaScript inheritance through prototype

```
// define the Person Class
function Person() {}
Person.prototype.walk = function(){
 console.log ('I am walking!');
};
Person.prototype.sayHello =
    function(){
 console.log ('hello');
};
// define the Student class
function Student() {}
// inherit Person
Student.prototype = new Person();
//modify the Person  prototype
Person.prototype.sing=function(){
 console.log("Rock and roll");
};
```

```
// replace the sayHello method
Student.prototype.sayHello = function(){
  console.log('hi, I am a student');
}

// add sayGoodBye method
Student.prototype.sayGoodBye = function(){
 console.log('goodBye');
}

  var student1 = new Student();
  student1.sayHello();
  student1.walk();
  student1.sayGoodBye();
  student1.sing();

  // check inheritance
  console.log(student1 instanceof Person);
  // true
  console.log(student1 instanceof Student);
  // true
```

So, what exactly is a prototype?

THE UNIVERSITY OF WESTERN AUSTRALIA

- A prototype is an object from which other objects inherit properties. Any object can be a prototype.
- Every object has a prototype by default. Since prototype are themselves objects, every prototype has a prototype too (There is only one exception, the default Object prototype at the top of every prototype chain).
- If you try to look up a key on an object and it is not found, JavaScript will look for it in the prototype. It will follow the prototype chain until it sees a null value. In that case, it returns undefined.

Setting object prototype

THE UNIVERSITY OF WESTERN AUSTRALIA

```
var man = Object.create(null);
defineProperty(man, 'sex', "male");

var yehuda = Object.create(man);
defineProperty(yehuda, 'firstName', "Yehuda");
defineProperty(yehuda, 'lastName', "Katz");

yehuda.sex          // "male"
yehuda.firstName    // "Yehuda"
yehuda.lastName     // "Katz"

Object.getPrototypeOf(yehuda)
                    // returns the man object
```

Inner functions

- JavaScript function declarations are allowed inside other functions

```
function betterExampleNeeded(){

    var a = 1;
    function oneMoreThanA(){
        return a + 1;
    }

     return oneMoreThanA();
}
```

- A closure is the local variables for a function – kept alive after the function has returned.

Why inner functions?

THE UNIVERSITY OF WESTERN AUSTRALIA

- This provides a great deal of utility in writing more maintainable code. If a function relies on one or two other functions that are not useful to any other part of your code, you can nest those utility functions inside the function that will be called from elsewhere. This keeps the number of functions that are in the global scope down, which is always a good thing.
- This is also a great counter to the lure of global variables. When writing complex code it is often tempting to use global variables to share values between multiple functions — which leads to code that is hard to maintain. Nested functions can share variables in their parent, so you can use that mechanism to couple functions together when it makes sense without polluting your global namespace — 'local globals' if you like. This technique should be used with caution, but it's a useful ability to have.

JavaScript Closure

THE UNIVERSITY OF WESTERN AUSTRALIA

- Using inner functions we can use one of the most powerful abstractions Javascript has to offer – closure. A quick quiz, what does this do –

```
function makeAdder(a) {
    return function(b){
        return a + b;
    }
}

x = makeAdder(5);
y = makeAdder(20);

console.log( x(6) );     // ?
console.log( y(7) );     // ?
```

Javascript closure (cont.)

- Here, the outer function (`makeAdder`) has returned, and hence common sense would seem to dictate that its **local variable** no longer exist. But they **do still exist**, otherwise the adder function would be unable to work.
- In actuality, whenever JavaScript executes a function, a **scope object** is created to hold the local variables created within that function. It is initialized with any variables passed in as function parameters.
- This is similar to the global object that all global variables and functions live in, but with a couple of important differences:
  - firstly, a brand new scope object is created every time a function starts executing, and
  - secondly, unlike the global object these scope objects cannot be directly accessed from your code.

Javascript closure (cont.)

- So when `makeAdder` is called, a scope object is created with one property: a, which is the argument passed to the function. It then returns a newly created function.
- Normally JavaScript's garbage collector would clean up the scope object created for `makeAdder` at this point, but the returned function maintains a reference back to that scope object. As a result, the scope object will not be garbage collected until there are no more references to the function object that makeAdder returned.
- Scope objects form a chain called the scope chain, similar to the prototype chain used by JavaScript's object system. **A closure is the combination of a function and the scope object in which it was created.** Closures let you save state — as such, they can often be used in place of objects.